Smart Officse Facility Programming Exercise

**Problem Statement**

Design a console-based application to manage a smart office facility. The system should handle conference room bookings, occupancy detection, and automate the control of air conditioning and lighting based on room occupancy. This exercise aims to evaluate the candidate's ability to implement best coding practices, design patterns, and create an efficient, maintainable, and scalable solution.

**Key Functional Requirements:**

1. Configure the office facility by specifying the number of meeting rooms.
2. Book and cancel bookings for conference rooms.
3. Detect occupancy using sensors when at least two people enter a room.
4. Automatically release bookings if the room is not occupied within 5 minutes.
5. Turn off air conditioning and lights if the room is not occupied.

**Key Focus:**

* **Singleton Pattern**: Ensure that the office configuration and room booking system is a single instance throughout the application.
* **Observer Pattern**: Implement sensors and control systems (lights, AC) as observers to the room's occupancy status.
* **Command Pattern**: Handle booking, cancellation, and room status updates through commands, allowing for flexible and extendable operations.

**Java Code Implementation**

**Singleton Pattern: OfficeConfiguration**

public class OfficeConfiguration {

private static OfficeConfiguration instance;

private List<ConferenceRoom> conferenceRooms;

private OfficeConfiguration() {

conferenceRooms = new ArrayList<>();

}

public static synchronized OfficeConfiguration getInstance() {

if (instance == null) {

instance = new OfficeConfiguration();

}

return instance;

}

public void configureRooms(int numberOfRooms) {

for (int i = 1; i <= numberOfRooms; i++) {

conferenceRooms.add(new ConferenceRoom(i));

}

System.out.println("Office configured with " + numberOfRooms + " meeting rooms.");

}

public List<ConferenceRoom> getConferenceRooms() {

return conferenceRooms;

}

}

**Observer Pattern: ConferenceRoom, OccupancySensor, Light, and AirConditioner**

import java.util.ArrayList;

import java.util.List;

public class ConferenceRoom {

private int roomNumber;

private boolean occupied;

private List<Observer> observers;

public ConferenceRoom(int roomNumber) {

this.roomNumber = roomNumber;

this.occupied = false;

this.observers = new ArrayList<>();

}

public void addObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void setOccupied(boolean occupied) {

this.occupied = occupied;

notifyObservers();

}

public boolean isOccupied() {

return occupied;

}

private void notifyObservers() {

for (Observer observer : observers) {

observer.update(this);

}

}

public int getRoomNumber() {

return roomNumber;

}

}

interface Observer {

void update(ConferenceRoom room);

}

public class OccupancySensor implements Observer {

@Override

public void update(ConferenceRoom room) {

if (room.isOccupied()) {

System.out.println("Room " + room.getRoomNumber() + " is now occupied.");

} else {

System.out.println("Room " + room.getRoomNumber() + " is now unoccupied.");

}

}

}

public class Light implements Observer {

@Override

public void update(ConferenceRoom room) {

if (room.isOccupied()) {

System.out.println("Lights turned on in Room " + room.getRoomNumber());

} else {

System.out.println("Lights turned off in Room " + room.getRoomNumber());

}

}

}

public class AirConditioner implements Observer {

@Override

public void update(ConferenceRoom room) {

if (room.isOccupied()) {

System.out.println("AC turned on in Room " + room.getRoomNumber());

} else {

System.out.println("AC turned off in Room " + room.getRoomNumber());

}

}

}

**Command Pattern: Command, BookingCommand, and CancelBookingCommand**

import java.util.ArrayList;

import java.util.List;

interface Command {

void execute();

}

public class BookingCommand implements Command {

private ConferenceRoom room;

public BookingCommand(ConferenceRoom room) {

this.room = room;

}

@Override

public void execute() {

if (!room.isOccupied()) {

room.setOccupied(true);

System.out.println("Room " + room.getRoomNumber() + " booked successfully.");

} else {

System.out.println("Room " + room.getRoomNumber() + " is already booked.");

}

}

}

public class CancelBookingCommand implements Command {

private ConferenceRoom room;

public CancelBookingCommand(ConferenceRoom room) {

this.room = room;

}

@Override

public void execute() {

if (room.isOccupied()) {

room.setOccupied(false);

System.out.println("Booking for Room " + room.getRoomNumber() + " cancelled successfully.");

} else {

System.out.println("Room " + room.getRoomNumber() + " is not booked.");

}

}

}

public class CommandInvoker {

private List<Command> commandHistory;

public CommandInvoker() {

commandHistory = new ArrayList<>();

}

public void executeCommand(Command command) {

command.execute();

commandHistory.add(command);

}

}

**Main Class to Run the Application**

public class SmartOfficeApp {

public static void main(String[] args) {

OfficeConfiguration office = OfficeConfiguration.getInstance();

office.configureRooms(3);

List<ConferenceRoom> rooms = office.getConferenceRooms();

ConferenceRoom room1 = rooms.get(0);

OccupancySensor sensor = new OccupancySensor();

Light light = new Light();

AirConditioner ac = new AirConditioner();

room1.addObserver(sensor);

room1.addObserver(light);

room1.addObserver(ac);

CommandInvoker invoker = new CommandInvoker();

Command bookRoom1 = new BookingCommand(room1);

Command cancelRoom1 = new CancelBookingCommand(room1);

invoker.executeCommand(bookRoom1);

invoker.executeCommand(cancelRoom1);

}

}

**Explanation**

1. **Singleton Pattern**: The OfficeConfiguration class ensures that only one instance of the office configuration exists.
2. **Observer Pattern**: ConferenceRoom notifies its observers (OccupancySensor, Light, and AirConditioner) whenever its occupancy status changes.
3. **Command Pattern**: The Command interface and its implementations (BookingCommand and CancelBookingCommand) handle booking and cancellation operations.